(4)

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-380

TYPE RECONSTRUCTION WITH FIRST-CLASS POLYMORPHIC VALUES

James W. O'Toole
David K. Gifford

May 1989

89   130

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
Unclassified	

2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TM-380	N00014-83-K-0125

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
MIT Laboratory for Computer Science		Office of Naval Research/Department of Navy

6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code)
545 Technology Square Cambridge, MA 02139	Information Systems Program Arlington, VA 22217

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
DARPA/DOD		

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
1400 Wilson Boulevard Arlington, VA 22217	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Type Reconstruction with First-Class Polymorphic Values

12. PERSONAL AUTHOR(S)
O'Toole, J.W. and Gifford, D.K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1989 May	11

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Languages, Type Theory, Polymorphism, type systems, effect systems, type inference, type reconstruction, FX-89

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present the first type reconstruction system which combines the implicit typing of ML with the full power of the explicitly typed second-order polymorphic lambda calculus. The system will accept ML-style programs, explicitly typed programs, and programs that use explicit types for all first-class polymorphic values. We accomplish this flexibility by providing both generic and explicitly-quantified polymorphic types, as well as operators which convert between thse two forms of polymorphism. This type reconstruction system is an integral part of the FX-89 programming language. We present a type reconstruction algorithm for the system. The type reconstruction algorithm is proven sound and complete with respect to the formal typing rules.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Judy Little, Publications Coordinator	(617) 253-5894	

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete

Type Reconstruction
with First-Class Polymorphic Values

James William O'Toole Jr.[*]
David K. Gifford[†]

Abstract

We present the first type reconstruction system which combines the implicit typing of ML with the full power of the explicitly typed second-order polymorphic lambda calculus. The system will accept ML-style programs, explicitly typed programs, and programs that use explicit types for all first-class polymorphic values. We accomplish this flexibility by providing both generic and explicitly-quantified polymorphic types, as well as operators which convert between these two forms of polymorphism. This type reconstruction system is an integral part of the FX-89 programming language. We present a type reconstruction algorithm for the system. The type reconstruction algorithm is proven sound and complete with respect to the formal typing rules.

Categories and Subject Descriptions: D.1.m [**Programming Techniques**] – Miscellaneous: *First-Class Polymorphism*; D.3.1 [**Programming Languages**] – Formal Definitions and Theory; D.3.3 [**Programming Languages**] – Language Constructs: *Implicit Typing*; D.3.4 [**Programming Languages**] – Processors: *Compilers*.

General Terms: Languages, Type Theory, Polymorphism.
Additional Key Words and Phrases: type systems, effect systems, type inference, type reconstruction, FX-89.

1 Combining Generic and First-Class Polymorphism

Type reconstruction relieves the programmer of the burden of providing type information while retaining the benefits of strongly-typed languages, including superior performance, documentation, and safety. However, present systems for type reconstruction, such as the ML type system [Milner78], do not permit the use of first-class polymorphic values. Explicitly-typed languages, such as FX-87 [Gifford87], do permit first-class polymorphic values, but they do not provide the programmer with the convenience of implicitly-typed languages such as ML.
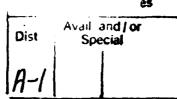
The FX-89 programming language is a revision and extension of FX-87. FX-89 is based on a type reconstruction system that combines the flexibility of ML with the full typing ability of the explicitly typed second order lambda calculus. This reconstruction system will accept ML-style programs, explicitly-typed programs, and programs that use explicit types for all first-class polymorphic values.

In this paper we describe both the theoretical basis of the FX-89 type reconstruction system and our type reconstruction algorithm. The algorithm described in the paper has been implemented.

In order to simplify our presentation we will restrict our attention to a simplified version of FX-89 which we will call IFX. The full FX-89 language includes side-effects, modules, oneofs, references, and other data types. These constructs can be added to the type inference system described below.

In the remainder of this paper we discuss the previous work in this area (Section 2), introduce IFX (Section 3), present a system of type reconstruction rules (Section 4), describe an algorithm which reconstructs IFX types (Section 5), prove the correctness of the algorithm (Section 6), briefly discuss possible extensions (Section 7), and conclude with some observations on our result (Section 8).

1

2 Previous Work

[Milner78] presents a typing system based on *type schemes* in which the **let** construct provides generic polymorphism. ML, as presented in [Damas82], uses generic type variables to express polymorphism. The ML type discipline is not as powerful as the type discipline of the second-order polymorphic lambda calculus [Fortune83]. Type quantifiers are not explicit in ML, and it is therefore not possible to express the type of a function which expects a polymorphic value as an argument. For this reason, we say that ML does not provide first-class polymorphism.

[McCracken84] introduced a type reconstruction system for the second-order polymorphic lambda calculus. McCracken's system did not provide the generic **let** construct of ML, although it did attempt to support the automatic instantiation of explicitly typed polymorphic functions in application position. Both [McCracken84] and [Leivant83] attempted to provide automatic type abstraction in more general type systems, but these results are flawed (see sections 4.5 and 4.6).

The general partial polymorphic type inference problem was shown to be undecidable by [Boehm85]. More recent work [Kfoury88] has shown that conservative extensions to ML providing restricted polymorphism are possible, but has not provided a practical type reconstruction algorithm. Recently, [Pfenning88] related the complexity of the partial type reconstruction problem for the second-order polymorphic lambda calculus to that of second-order unification, which is well-known to be undecidable [Goldfarb81].

We believe our type reconstruction method is the first to combine the implicit typing of ML with the full power of the second-order polymorphic lambda calculus. We accomplish this flexibility by providing both generic and explicitly-quantified polymorphic types, as well as operators which convert between these two forms of polymorphism. The difficulty of second-order unification is avoided via syntactic restrictions that define the types which may be omitted by the programmer.

Below are some examples which illustrate the relative power of the ML typing system, McCracken's typing system (MTS), and our system (IFX). The **let** binding construct of ML permits generic type abstraction and instantiation of **twice** and **id**:

```
(let ((twice (lambda (f x) (f (f x))))
      (id (lambda (x) x)))
  (cons (twice id 0) (twice id true)))
```

This example is not well-typed in MTS because no generic polymorphism is provided. This example is well-typed in ML because the variables **twice** and **id** are assigned generic types, and these generic types are automatically instantiated as necessary. In general, ML programs cannot be typed by MTS without the addition of extensive explicit type abstraction and instantiation information. The explicit typing of MTS, and the implicit instantiation of the functions **cons** and **f**, permit:

```
(lambda (f :  ∀t.t → t)
  (cons (f 0) (f true)))
```

The second example is not well-typed in ML, because the lambda-bound variable **f** must have two incompatible types within the body of the lambda. In ML, A lambda-bound variable cannot be given a generic type within the body of the lambda because the type language is not capable of expressing the resulting function type of the lambda, which must contain an explicit type quantifier. First-class polymorphism allows the variable **f** to be assigned an explicitly quantified polymorphic type.

Our system permits both of the above examples. McCracken introduced the **close** operator to allow the programmer to indicate where type abstraction should occur without having to specify precisely what those type abstractions should be. A discussion of the formal typing rule for the **close** operator and why our modification to the rule in [McCracken84] is necessary appears in sections 4.5 and 4.6.

Unlike McCracken's system, our type system contains both ML-style generic types and explicitly quantified types, and we therefore require that the programmer indicate where explicit quantifiers should be removed from a type. The following example illustrates the use of both explicit and generic polymorphism:

```
(lambda (g :  ∀t.t → t)
  (let ((twice (lambda (f x)
                 (f (f x)))))
    (g (open g)))
  (cons (twice g 0)
        (twice g true))))
```

The **open** operator is used to convert an explicitly quantified polymorphic type into an ML-style generic polymorphic type, and **close** is used to make the opposite conversion.

2

In sum, the major advantages of our type reconstruction system are:

- Programmers may write without using explicit type specifications. ML-style programs may be used without modification in our type system.

- Programmers may write using fully explicit type specifications. Explicitly-typed programs may be used without modification in our type system.

- Programmers may use explicit types where desirable for documentation or other purposes, and omit them where they would decrease readability.

- First-class polymorphic values can be used, provided that their types are declared. Thus, modules can be first-class values in our system. Programmers may use the open and close operators to simplify the use of first-class polymorphic values.

3 IFX: A Typed Language

For pedagogical purposes we will study type reconstruction for IFX, a simplified version of FX-89. FX-89 is a polymorphic typed language that allows side-effects and first-class functions. Its syntax and most of its standard operations are strongly inspired by Scheme [Rees86]. The language IFX has the following Type and Expression domains (where I is the domain of identifiers, and π_i are the primitive types):

ι	:	I	::=	Identifiers
π	:	P	::=	Primitive types

υ	:	U	::=	P	primitive type
				I	type identifier
				U → U	function
				$\forall I_1...I_n.U$	polymorphic type

e	:	E	::=	I	variable
				(lambda (I) E)	lambda
				(lambda (I : U) E)	abstraction
				(E E)	application
				(let (I E) E)	generic-let
				(plambda (I...) E)	type abstraction
				(close E)	type closure
				(proj E U...)	projection
				(open E)	automatic projection

The type domain U contains the types which are supplied by the programmer in explicit type declarations.

The type of a function encodes the type of its argument and its result. If the type of the argument is monomorphic, then it may be omitted. The type $\forall t.v$ represents the type of polymorphic values abstracted over the type parameter t.

In the expression domain, just as **lambda abstracts** E over the ordinary variable I of type U, **plambda** abstracts E over the type identifier I to yield a polymorphic value. A polymorphic value is instantiated with the **proj** construct. The **close** and **open** constructs provide automatic type abstraction and instantiation. As an example, we give below the code of the polymorphic **compose** function that composes the function **f** with the function **g**:

```
(close
  (lambda (f)
    (lambda (g)
      (lambda (x)
        (g (f x)))))))
```

Note that **compose** is automatically abstracted over the argument and result types of the functions **f** and **g**. The type of **compose** is:

$$\text{compose} : \forall tsr.(t \to s) \to (s \to r) \to t \to r$$

The types of **f**, **g**, and **x** were omitted from the program text, but could equally well have been included by the programmer.

4 Type Reconstruction System

We present the typing system of IFX as as formal deduction system consisting of a set of type reconstruction rules. The type system contains *generic* (i.e. general) type variables, and distinguishes between these generic type variables and the type identifiers which appear in user-supplied types. The type system also distinguishes between monomorphic and polymorphic types:

α	:	G	::=		General type variables

μ	:	M	::=	P	primitive type
				I	type identifier
				G	general type variable
				M → M	function

τ	:	T	::=	P	primitive type
				I	type identifier
				G	general type variable
				T → T	function
				$\forall I_1...I_n.T$	polymorphic type

The IFX typing rules make use of an important distinction between the M and T type domains. The rules are designed so that M types may be omitted from formal argument type declarations, but T types may not. Thus, the different levels in our type syntax specify the restrictions on the input programs. The use of syntactically-specified restrictions is intended to communicate clearly to the programmer the limitations of the type reconstruction system.

4.1 Type Schemes

The IFX type system supports the generic polymorphism found in ML, as well as the explicit polymorphism found in Reynolds' second-order polymorphic lambda calculus. In order to provide generic polymorphism, we define *type schemes*, which represent the *generic* (i.e. general) type of a variable which is permitted multiple *instantiations*:

Definition (Type Scheme). A *type scheme* η is a term of the form

$$\tilde{\forall}\alpha_1...\alpha_n.\tau,$$

where $\alpha_1...\alpha_n$ are the *generic variables* of $\tau \in$ T.

We distinguish $\tilde{\forall}$ and $\forall$ deliberately: $\tilde{\forall}$ binds the generic type variables of a type scheme, and $\forall$ binds type variables within a type.

Definition (Alpha-renaming). Types τ and τ' are alpha-renamable (written $\tau \simeq \tau'$) iff some renaming of type variables bound in τ produces τ'.

Definition (Instantiation). The type τ' is an instance of the scheme $\eta = \tilde{\forall}\alpha_1...\alpha_n.\tau$ (written $\eta \succeq \tau'$) iff there are monomorphic $\mu_1...\mu_n$ such that $\tau[\mu_i/\alpha_i] \simeq \tau'$. (We extend $\succeq$ to type schemes by $\eta \succeq \eta'$ iff $\forall \tau : \eta' \succeq \tau \Rightarrow \eta \succeq \tau$.)

Note that only M types may be substituted to produce instantiations, and that we assume that substitution takes place with renaming of any bound type variables to avoid capture. The result of substituting μ for t in τ will be written $\tau[\mu/t]$. The type scheme $\eta = \tilde{\forall}.\tau$, having no generic type variables, will occasionally be abbreviated as τ.

We first present the inference rules for explicitly typed terms. A type assignment A maps each variable in its domain to a type scheme. We will use A_x to refer to the type assignment A with the assignment for variable x removed.

We use FGV(τ) to refer to the free *general type variables* of τ, and FTV(τ) to refer to the free *type identifiers* of τ. Similarly, FGV(A) refers to the free *general type variables* of the type schemes in the assignment A. We define Gen(A, τ), as follows:

Definition (Generalization). The *generalization* of τ with respect to A (written Gen(A, τ)), is the type scheme $\eta = \tilde{\forall}\alpha_i.\tau$, where $\{\alpha_i\} = $ FGV(τ) $-$ FGV(A).

4.2 The Deduction System

The type reconstruction rules of IFX are as follows:

LAMBDA

$$\frac{A_x + (x : \upsilon) \vdash e : \tau}{A \vdash (\texttt{lambda } (x : \upsilon) \ e) : \upsilon \to \tau}$$

APPL

$$\frac{A \vdash e : \tau_a \to \tau_r \qquad A \vdash e_a : \tau_a}{A \vdash (e \ e_a) : \tau_r}$$

PLAMBDA

$$\frac{A \vdash e : \tau \qquad A \vdash e : \tau' \Rightarrow \text{Gen}(A, \tau) \succeq \text{Gen}(A, \tau') \qquad t_i \notin \text{FTV}(A)}{A \vdash (\texttt{plambda } (t_i) \ e) : \forall t_i.\tau}$$

PROJ

$$\frac{A \vdash e : \forall t_i.\tau}{A \vdash (\texttt{proj } e \ \upsilon_i) : \tau[\upsilon_i/t_i]}$$

The above rules describe the typing requirements of value abstraction, value application, type abstraction, and type application.

The following rules describe the typing requirements of our **open/close** operators, ML-style generic **let**, and the automatic type application of polymorphic functions. These typing rules permit convenient use of polymorphic values. Note that only M types are synthesized by the reconstruction rules, as the omitted types of **lambda**-bound variables, or as type parameters in **open** or generic variable instantiation. The T types, containing explicit quantifiers, must be provided explicitly.

OPEN

$$\frac{A \vdash e : \forall t_i.\tau}{A \vdash (\texttt{open } e) : \tau[\mu_i/t_i]}$$

4

CLOSE

$$A \vdash e : \tau$$
$$A \vdash e : \tau' \Rightarrow \mathrm{Gen}(A, \tau) \succeq \mathrm{Gen}(A, \tau')$$
$$\{\alpha_i\} = \mathrm{FG\tilde{V}}(\tau) - \mathrm{FGV}(A)$$
$$\overline{A \vdash (\texttt{close } e) : \forall \alpha_i . \tau}$$

VARINST

$$(x : \tilde{\forall} \alpha_i . \tau) \in A$$
$$\overline{A \vdash x : \tau[\mu_i / \alpha_i]}$$

ILET

$$A \vdash e_b : \tau_b$$
$$A \vdash e_b : \tau_b' \Rightarrow \mathrm{Gen}(A, \tau_b) \succeq \mathrm{Gen}(A, \tau_b')$$
$$A_x + (x : \mathrm{Gen}(A, \tau_b)) \vdash e : \tau$$
$$\overline{A \vdash (\texttt{let } (x \; e_b) \; e) : \tau}$$

ILAMBDA

$$A_x + (x : \mu) \vdash e : \tau$$
$$\overline{A \vdash (\texttt{lambda } (x) \; e) : \mu \to \tau}$$

IAPPL

$$A \vdash e : \forall t_i . (\tau_f \to \tau_r)$$
$$A \vdash e_a : \tau_a$$
$$\tau_a = \tau_f[\mu_i / t_i]$$
$$\overline{A \vdash (e \; e_a) : \tau_r[\mu_i / t_i]}$$

4.3 Generic let

The ILET and VARINST rules provide the ML-style generic `let`. ILET associates a generic type scheme with the `let`-bound variable, and VARINST permits each occurence of the variable to be independently assigned any instance of its generic type scheme. The convenience of automatic generalization and instantiation are provided by these two rules. In IFX, the typing rules permit this convenience with the caveat that the automatically deduced type parameters be M types.

The typing power of the ILET rule is equivalent to that provided by rewriting the `let` expression in the usual way, while making use of `open` and `close`:

$$((\texttt{lambda } (x : \tau) \; e[(\texttt{open } x)/x]) \; (\texttt{close } e_b)).$$

However, this transformation is not pure syntactic sugar, because it requires τ, the explicitly polymorphic type of the bound variable.

4.4 Implicit Projection

The IAPPL rule illustrates how implicit type application of polymorphic functions may be provided. The implicit instantiation is achieved by making use of the `open` operator to obtain a generic type. The application $(e \; e_a)$ is typed by IAPPL as if written $((\texttt{open } e) \; e_a)$.

4.5 Discussion of the CLOSE Rule

The `open` and `close` operators provide the programmer with the means to convert types between the explicitly quantified style and the generic polymorphism style. The `open` operator converts an explicitly quantified polymorphic type into one of its instantiations. Using `open` can be understood as requesting a type application to automatically determined M type parameters. Therefore, `open` is more convenient to use than `proj`, but `proj` must be used when application to T types is desired.

The `close` operator takes an expression whose type contains unbound general type variables, and performs type abstraction with respect to those type variables. The IFX operator `close` has the same intended semantics as McCracken's type closure operator, with one important difference: the IFX `close` operator acts *only on the most general type of the expression*. This restriction is enforced by the CLOSE rule, which contains as an antecedent that the type used for the closure be the most general type of the expression. This restriction was not included in [McCracken84], and precludes the completeness of the typechecking algorithm W presented in that work.

4.6 Only the Most General Type

The difficulty with the CLOSE rule (and similarly PLAMBDA and ILET), is that in the absence of the most general type restriction, a more specific type may be chosen, leading to a difference in the form of the explicitly quantified polymorphic type. Consider the following simple example:

$$(\texttt{proj } (\texttt{close } (\texttt{lambda } (x) \; x)) \; s \; r).$$

The algorithm given in [McCracken84], and our algorithm, will fail on the above example. The reason is that the natural typing of $(\texttt{close } (\texttt{lambda } (x) \; x))$ is $\forall t . t \to t$, which cannot be applied to two types (s and r), but only to a single type. Without the most general type restriction in the CLOSE rule, the rule would indicate that we may instead deduce the type $(t_1 \to t_2) \to t_1 \to t_2$ for $(\texttt{lambda } (x) \; x)$. This type would be closed to produce $\forall t_1 \; t_2 . (t_1 \to t_2) \to t_1 \to t_2$, which could be applied to the two types s and r.

The CLOSE typing rule in IFX imposes the requirement that the type closed be the most general type of the body. This antecedent solves the problem described above. Also, for purposes of disambiguation, the type variables are bound in the same order in which they appear in the type of the expression. These conditions

5

ensure that the explicitly polymorphic type of a close expression is "frozen," and will have a fixed polymorphic structure which:

- Can be relied upon by the programmer; so that there is no ambiguity as to the form of the type.

- Avoids the need to extend the instantiation relation to structurally different polymorphic types.

- Allows the type reconstruction algorithm to compute the unique most general type when processing a close expression.

This last item guarantees that the algorithm need make no arbitrary choices thus avoiding the need for backtracking.

5 A Type Reconstruction Algorithm

The type reconstruction algorithm, R, computes types for IFX expressions that are consistent with the deductive rule system in section 4. Algorithm R manipulates type expressions which contain unification variables. We have chosen to represent unification variables explicitly, to avoid any possible confusion regarding the status of particular type variables occurring within type expressions. To summarize the notation:

t Explicitly quantified, bound type variable.

α Generic type variable, implicitly bound.

μ Monomorphic type, possibly generic, omittable.

τ Unrestricted type.

ν Unification variable, which represents an M type being reconstructed.

Because a ν represents an M type, it cannot be unified with type expressions containing $\forall$.

Our type reconstruction algorithm makes extensive use of a unification algorithm for this type system. Our unification algorithm and its implementation are based on the work of [Morris68,Hindley69]. As in [McCracken84], we require that the unification algorithm be extended to support alpha-renaming of bound type variables, including generic type variables. To state the substitution lemma correctly, we must prevent the possibility of user type identifier capture, so we make use of BTV(e), the type identifiers bound by plambda within

e. We say that S is a *valid* M-substitution for A and e if S maps type variables in FGV(A) to M types which do not contain any BTV(E).

Lemma (Substitution). Given a type assignment A, expression e, type τ, and a valid M-substitution S:

$$A \vdash e : \tau \Rightarrow SA \vdash e : S\tau$$

Proof: By structural induction on e. □

Lemma (Extended Unification). There exists an algorithm $U(\tau, \tau')$ with these properties:

- Supports alpha-renaming of bound type variables, including generic variables.

- Matches unification variables only with M types.

- Returns an M-substitution unifying τ and τ' or fails, according to the usual unification rule defined by $\succeq$.

If there exists an M-substitution unifying τ and τ', then $U(\tau, \tau')$ will return the most general unifying substitution:

$$S\tau \simeq S\tau'$$
$$\Rightarrow$$
$$U(\tau, \tau') \to \hat{S}$$
$$\hat{S}\tau \simeq \hat{S}\tau'$$
$$\exists P : (P\hat{S} \simeq S),$$

and otherwise $U(\tau, \tau')$ will fail:

$$\forall S : S\tau \not\simeq S\tau'$$
$$\Rightarrow$$
$$U(\tau, \tau') \to failure.$$

Proof: Straightforward. □

Algorithm R. The reconstruction algorithm is defined as a recursive procedure. Algorithm $R(A, e)$ takes the type assignment A and the input expression e, and computes $\langle S, \tau \rangle$. The algorithm fails if any of the invocations of the unification algorithm fail, if any of the recursive invocations of R fail, or if failure is specifically indicated in the algorithm text, given below. The substitution S is an M-substitution on FGV(A). The substitution incorporates any information about the types of the variables in $dom(A)$ which is discovered by traversing the expression e. The design is such that τ is the most general type of e under the assignment SA, and S is the minimal substitution on FGV(A) which permits e to be typed.

6

$R(A, e) \equiv$ **case** e **of**:

x

    if $(x : \tilde{\forall}\alpha_i.\tau) \notin A$ then fail

    return $\langle \phi, \tau[\nu_i/\alpha_i]\rangle$

        where ν_i are fresh.

(proj e ν_i)

    let $\langle S, \tau \rangle = R(A, e)$

    if $\tau \neq \forall t_i.\tau'$ then fail

    return $\langle S, \tau'[\nu_i/t_i]\rangle$.

(plambda (t_i) e)

    let $\langle S, \tau \rangle = R(A, e)$

    if any $t_i \in \text{FTV}(SA)$ then fail

    return $\langle S, \forall t_i.\tau \rangle$.

(lambda $(x : \upsilon)$ e)

    let $\langle S, \tau_r \rangle = R(A + (x : \upsilon), e)$

    return $\langle S, \upsilon \to \tau_r \rangle$.

(lambda (x) e)

    let $\langle S, \tau \rangle = R(A + (x : \nu), e)$

        where ν is fresh

    return $\langle S, S\nu \to \tau \rangle$.

(e e_a)

    let $\langle S, \tau_m \rangle = R(A, e)$

    let $\langle S_a, \tau_a \rangle = R(SA, e_a)$

    if $\tau_m = \forall t_i.\tau'$ then

        let $\tau = \tau'[\nu_i/t_i]$ where ν_i are fresh

    else

        let $\tau = \tau_m$

    if $\tau \neq \tau_f \to \tau_r$ then fail

    let $V = U(S_a\tau_f, \tau_a)$

    return $\langle VS_aS, VS_a\tau_r \rangle$.

(open e)

    let $\langle S, \tau \rangle = R(A, e)$

    if $\tau \neq \forall t_i.\tau'$ then fail

    return $\langle S, \tau'[\nu_i/t_i]\rangle$

        where ν_i are fresh.

(close e)

    let $\langle S, \tau \rangle = R(A, e)$

    let $\{\alpha_i\} = \text{FGV}(\tau) - \text{FGV}(SA)$

    return $\langle S, \forall \alpha_i.\tau \rangle$.

(let $(x$ $e_b)$ e)

    let $\langle S_b, \tau_b \rangle = R(A, e_b)$

    let $\{\alpha_i\} = \text{FGV}(\tau_b) - \text{FGV}(S_bA)$

    let $\langle S, \tau \rangle = R(S_bA + (x : \tilde{\forall}\alpha_i.\tau_b), e)$

    return $\langle SS_b, \tau \rangle$.

endcase.

6 Formal Properties of the Reconstruction Algorithm

An algorithm is a type reconstruction algorithm for IFX iff the algorithm has certain properties: *termination*, *soundness*, and *completeness*. Each of these properties may be viewed as a guarantee to the programmer about the behavior of the algorithm:

- The algorithm will always halt, either failing, or providing a type of the program.

- If the algorithm computes a type for the program, then the program can be proved to have that type according to the formal rules of IFX.

- If the program has some type according to the formal rules of IFX, then the algorithm will compute the most general type of the program.

We prove that Algorithm R has each of these three properties. For convenience, we will write $R(A, e) \to \langle S, \tau \rangle$ when we mean "Algorithm R halts on input A and e, yielding S and τ."

6.1 Termination of Algorithm R

Theorem (Termination). *Algorithm R terminates on all (finite) inputs.*

Proof: Observe that Algorithm R is a syntax-directed algorithm. $R(A, e)$ recursively traverses the syntax of the input expression e, which is finite. Algorithm R invokes the unification algorithm finitely many times, and each invocation must terminate, because all type expressions in IFX are finite. $\quad\square$

6.2 Correctness of Algorithm R

The notions of soundness and completeness of a typing algorithm with respect to a formal typing system are well-known. The most general type restriction in the antecedents of several of the IFX typing rules complicate the proofs of the soundness and completeness of R. Specifically, the soundness of R and the completeness of R cannot be proved independently, as is usually done [Damas85]. Consider, for example, the expression (close e). R on (close e) cannot be sound unless the type computed by R for e satisfies the most general type restriction, which means R must be complete. Similarly, the soundness of R is used to prove its completeness.

The argument is not circular, but mutually recursive. In other words, the soundness and completeness of R must be proved together, by structural induction.

6.3 Soundness of Algorithm R

We show that Algorithm R is sound with respect to the formal typing system in this sense:

Theorem (Syntactic Soundness). Given any type assignment A and expression e, if Algorithm R on input A and e computes the substitution S and the type τ, then e has type τ under the assignment SA:

$$R(A, e) \rightarrow \langle S, \tau \rangle$$

$$\Rightarrow$$

$$SA \vdash e : \tau.$$

Proof sketch: By structural induction on e, in tandem with the proof of completeness. Assume, by induction, that R is both sound and complete when applied to the component expressions of e. For each possible case of Algorithm R, there is a corresponding inference rule in IFX. In each case, the antecedents of the inference rule follow from the inductive hypothesis and the unification lemma. □

6.4 Completeness of Algorithm R

We show that Algorithm R is complete with respect to the formal typing system in the following sense:

Theorem (Syntactic Completeness). If e has type τ under assignment SA, where S is a valid M-substitution for A and e, then $R(A, e)$ will compute a substitution $\hat{S}$ and a type $\hat{\tau}$, such that $\hat{S}A$ is more general than SA, and $\hat{\tau}$ is more general than τ:

$$SA \vdash e : \tau$$

$$\Rightarrow$$

$$R(A, e) \rightarrow \langle \hat{S}, \hat{\tau} \rangle$$

$$\exists P : (P\hat{S}A \simeq SA) \wedge (P\mathrm{Gen}(\hat{S}A, \hat{\tau}) \succeq \mathrm{Gen}(SA, \tau)).$$

Proof sketch: By structural induction on e, in tandem with the proof of soundness. Assume, by induction, that R is both sound and complete when applied to the component expressions of e. Given that $SA \vdash e : \tau$, the final step in the formal deduction uses a typing rule of IFX. For each possible typing rule, there is a

corresponding case in Algorithm R. In each case, the antecedents of the typing rule permit the use of the induction hypothesis. The behavior of R on input e then follows from the application of the inductive hypothesis to the recursive calls to R on component expressions of e, and the unification lemma.

The detailed proof of the inductive step for (close e) illustrates the use of the induction hypothesis, the connection with the tandem proof of soundness, and the importance of the most general type restriction as an antecedent of the CLOSE rule:

Case CLOSE. Given that

$$SA \vdash (\text{close } e) : \forall \alpha_i.\tau, \tag{1}$$

we will show:

$$R(A, (\text{close } e)) \rightarrow \langle \hat{S}, \forall \alpha_j.\hat{\tau} \rangle, \tag{2}$$

and demonstate the existence of a substitution P' such that:

$$P'\hat{S}A \simeq SA \tag{3}$$

$$P'\mathrm{Gen}(\hat{S}A, \forall \alpha_j.\hat{\tau}) \succeq \mathrm{Gen}(SA, \forall \alpha_i.\tau). \tag{4}$$

The IFX deduction rule for (close e) proving (1) is CLOSE, so therefore

$$SA \vdash e : \tau \tag{5}$$

$$\{\alpha_i\} = \mathrm{FGV}(\tau) - \mathrm{FGV}(SA) \tag{6}$$

$$SA \vdash e : \tau' \Rightarrow \mathrm{Gen}(SA, \tau) \succeq \mathrm{Gen}(SA, \tau'). \tag{7}$$

By the inductive hypothesis on (5), there exists a substitution P such that:

$$R(A, e) \rightarrow \langle \hat{S}, \hat{\tau} \rangle \tag{8}$$

$$P\hat{S}A \simeq SA \tag{9}$$

$$P\mathrm{Gen}(\hat{S}A, \hat{\tau}) \succeq \mathrm{Gen}(SA, \tau) \tag{10}$$

$$\hat{S}A \vdash e : \hat{\tau} \tag{11}$$

In order to show (4), we will use the most general type restriction to show that the two type schemes in (10) are $\simeq$. By the substitution lemma, it follows from (9) and (11) that

$$SA \simeq P\hat{S}A \vdash e : P\hat{\tau} \tag{12}$$

But τ satisfies the most general type restriction (7), so the above implies

$$\mathrm{Gen}(SA, \tau) \succeq \mathrm{Gen}(SA, P\hat{\tau}). \tag{13}$$

8

Combining (10) and (13) we have

$$PGen(\hat{S}A, \hat{\tau}) \succeq Gen(SA, \tau) \succeq Gen(SA, P\hat{\tau}). \quad (14)$$

Now $dom(P) \subseteq FGV(\hat{S}A)$, so by definition of Gen, it is clear that

$$PGen(\hat{S}A, \hat{\tau}) \simeq Gen(P\hat{S}A, P\hat{\tau}). \quad (15)$$

Thus, by (9) and (15), the middle of (14) is excluded, so that the type schemes of (14) are all $\simeq$, specifically

$$PGen(\hat{S}A, \hat{\tau}) \simeq Gen(SA, \tau). \quad (16)$$

We must show that Algorithm R on (close e) computes a substitution and a type having the properties specified in the theorem. By definition of R on (close e), it follows from (8) that

$$R(A, (\text{close } e)) \rightarrow \langle \hat{S}, \forall \alpha_j.\hat{\tau} \rangle, \quad (17)$$

where $\{\alpha_j\} = FGV(\hat{\tau}) - FGV(\hat{S}A)$. We must also find a substitution P' such that:

$$P'\hat{S}A \simeq SA \quad (18)$$

$$P'Gen(\hat{S}A, \forall \alpha_j.\hat{\tau}) \succeq Gen(SA, \forall \alpha_i.\tau). \quad (19)$$

We choose $P' = P$ and observe that (18) follows from (9). By the choice of $\{\alpha_i\}$ and $\{\alpha_j\}$, it is obvious that the Gen operations in (19) do not bind any generic type variables. Therefore, (19) reduces to (16), by the definition of $\succeq$. We have shown (2), (3), and (4), and this completes the proof. □

7 Possible Extensions

The typing system of IFX may be extended to provide a richer type language. For example, the FX-89 design includes static side-effect information, module values, the usual sum and product types, and recursive definitions. Various extensions are briefly discussed.

7.1 Recursive Definitions

An important omission from the IFX typing rules is any means to define values recursively. The following two rules provide both implicitly and explicitly typed versions of letrec:

LETREC

$$\frac{A_x + (x : v_b) \vdash e_b : v_b \qquad A_x + (x : v_b) \vdash e : \tau}{A \vdash (\text{letrec } (x : v_b \ e_b) \ e) : \tau}$$

ILETREC

$$\frac{A_x + (x : \mu) \vdash e_b : \mu \qquad A_x + (x : \mu') \vdash e_b : \mu' \Rightarrow Gen(A_x, \mu) \succeq Gen(A_x, \mu') \qquad A_x + (x : Gen(A_x, \mu)) \vdash e : \tau}{A \vdash (\text{letrec } (x \ e_b) \ e) : \tau}$$

The LETREC rule requires that the programmer specify the type, but provides no generic polymorphism. The ILETREC rule provides generic polymorphism, but does not permit T types for the bound variable. Neither rule permits the bound variable to be generic within its own defining expression. We believe that algorithm R extends to these rules in a straightforward manner.

The following rule combines the good features of the above rules:

LR2

$$\frac{A_x + (x : \tau_b) \vdash e_b : \tau_b \qquad A_x + (x : \tau_b') \vdash e_b : \tau_b' \Rightarrow Gen(A_x, \tau_b) \succeq Gen(A_x, \tau_b') \qquad A_x + (x : Gen(A_x, \tau_b)) \vdash e : \tau}{A \vdash (\text{letrec } (x \ e_b) \ e) : \tau}$$

The LR2 rule permits the omission of the T type of the bound variable. We have not extended Algorithm R to compute types consistent with rule LR2. Whether such types can be computed is an open question.

7.2 Implicit Projection

The version of our IAPPL rule presented in [Mc-Cracken84] does not restrict the type arguments to be M types, but rather requires that all type arguments appear in the types of the formal subroutine arguments:

IPA

$$\frac{A \vdash e : \forall t_i.(\tau_f \rightarrow \tau_r) \qquad A \vdash e_a : \tau_a \qquad \{t_i\} \subseteq FTV(\tau_f) \qquad \tau_a = \tau_f[\tau_i/t_i]}{A \vdash (e \ e_a) : \tau_r[\tau_i/t_i]}$$

This extension corresponds to a straightforward modification to algorithm R and the unification algorithm. However, the completeness of the resulting algorithm remains an open question.

9

7.3 Type Closure w.r.t. User Type Identifiers

The CLOSE rule of IFX does not permit closure with respect to type variables which appear in user-supplied types. For this reason, the type variables which the programmer chooses to write explicitly must also be abstracted explicitly, via plambda. For example, (close (lambda (x, x)) : $\forall t.t \rightarrow t$, but we do not have (close (lambda ($x : t$) x)) : $\forall t.t \rightarrow t$ and instead must write (plambda (t) (lambda ($x : t$) x)).

The type reconstruction algorithm R can be modified to perform type closures with respect to these user-supplied type identifiers, and the resulting flexibility may be desirable. Unfortunately, we have not found a typing rule for close which both permits (close (lambda ($x : t$) x)) : $\forall t.t \rightarrow t$ and prevents

```
(lambda (y)
  (close (lambda (x)
    (lambda (w : t)
      ((lambda (z) x) y)))))
:  t → ∀s.(s → t → s)
```

The difficulty is that if the rule for close does permit abstraction with respect to user-supplied type identifiers, then the behavior of close depends upon the choice of omitted M types on lambda-bound variables. Particular choices of omitted M types will enlarge $FTV(A)$, thus preventing some type abstractions.

In the above example, if y is assigned the type t, then t is not abstractable by the close appearing in the body, because it appears free in the type assignment. However, the modified algorithm R would not assign y the type t, because the type of y is not constrained by the example. Therefore, closure with respect to t would be performed. This problem does not occur with generic type variables, because they cannot appear in the types supplied by the programmer. This may be seen by comparing the definitions of the type domains U and T.

7.4 Effect specifications

The type language of FX-87 provides *side-effect specifications*, which permit a fine-grained description of allocation, access, and mutation actions performed during the evaluation of an expression. These effect specifications are attached to function types, in order to permit accurate checking at the time of compilation.

The IFX typing rules permit monomorphic types to be omitted from formal argument type declarations. Ef-

fect specifications can be included in such a classification; such an approach is adopted by the current design of FX-89.

7.5 Applicative Types

In FX-87, type abstraction is permitted only when the side-effect specifications ensure that the polymorphic expression is referentially transparent. [Tofte87] takes a different approach, based on the concept of *applicative types*. Tofte classifies certain expressions as *expansive*, and permits type abstraction of these expressions only with respect to *applicative* type variables. This type abstraction rule permits different type abstractions than does the FX-87 pure-plambda rule.

A thorough comparison of these two abstraction rules is beyond the scope of this paper. [Tofte87] treats the issue of type abstraction in the presence of mutable data. We believe that the imperative typing discipline introduced in [Tofte87] could be combined with the type reconstruction system we have presented.

7.6 IFX as a Design Point

The language IFX may be viewed as one point in a design space of languages. The rules comprising IFX provide certain language constructs which define the behavior and interactions of generic and explicitly polymorphic types. However, other typing rules are also possible. For example, the VARINST rule could be changed to (in effect) automatically open any lambda-bound variable, except where some syntactic device appears.

The particular rules chosen will influence the programmer's use of polymorphism, as some usages are made more convenient than others. Choosing among such design alternatives is an engineering decision which may require empirical investigation.

8 Conclusion

We have developed a type reconstruction system which combines the convenience of ML-style typing with the full explicit typing power of the second-order polymorphic lambda calculus. The system permits the coexistence in the type domain of generic and explicitly polymorphic types, and provides operators with which the programmer may control conversions between the two forms of polymorphism. We have introduced the "most

general type" restriction for type abstractions, permitting the use of a unification-based typing algorithm. A practical type reconstruction algorithm for this language has been exhibited and its correctness has been proven.

Acknowledgments

References

[Boehm85] Boehm, H. J., "Partial polymorphic type inference is undecidable", *26th Annual Symposium on Foundations of Computer Science*, October 1985, pages 339–345.

[Damas82] Damas, L., Milner, R., "Principal type-schemes for functional programs", *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, January 1982, pages 207–212.

[Damas85] Damas, L., "Type Assignment in Programming Lanuages", Ph.D. Thesis CST-33-85, University of Edinburgh, April 1985.

[Fortune83] Fortune S., Leivant D., and O'Donnell M., "Simple and Second-order Type Structures", *Journal of the Association for Computing Machinery*, January 1983.

[Gifford87] Gifford, D. K., Jouvelot, P., Lucassen, J. M., Sheldon, M. A., *The FX-87 Reference Manual*, MIT/LCS/TR-407, October 1987.

[Goldfarb81] Goldfarb, W. D., "The undecidability of the second-order unification problem", *Theoretical Computer Science*, vol. 13, 1981, pages 225–230.

[Hindley69] Hindley, R., "The principal type-scheme of an object in combinatory logic", *Transactions of the American Mathematical Society*, vol. 146, 1969, pages 29–60.

[Kfoury88] Kfoury, A. J., Tiuryn, J., and Urzyczyn, P., "A Proper Extension of ML with an effective Type-Assignment", *Proceedings of the 15th Annual POPL*, 1988.

[Leivant83] Leivant, D., "Polymorphic Type Inference", *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, 1983.

[Lucassen87] Lucassen, J. M., *Types and Effects: Towards the Integration of Functional and Imperative Programming*, Ph.D. Thesis MIT/LCS/TR-408, Massachusetts Institute of Technology, September 1987.

[MacQueen84] MacQueen, D., "Modules for Standard ML", *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, 1984, pages 198–207.

[McCracken79] McCracken, N. J. *An Investigation of a Programming Language with a Polymorphic Type Structure*. Ph.D. Thesis, Syracuse University, 1979.

[McCracken84] McCracken, N. J., "The Typechecking of Programs with Implicit Type Structure", *Semantics of Data Types*, vol. 173 of Lecture Notes in Computer Science, June 1984, pages 301-315.

[Milner78] Milner, R., "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, vol. 17, 1978, pages 349–375.

[Morris68] Morris, J. H., *Lambda-Calculus Models of Programming Languages*, Massachusetts Institute of Technology, MAC-TR-57, 1968.

[O'Toole89] O'Toole, James William, Jr., *Type Reconstruction with First Class Polymorphic Values*, MIT/LCS/TM-380, 1989.

[Pfenning88] Pfenning, Frank, "Partial Polymorphic Type Inference and Higher Order Unification", *Proceedings of ACM Conference on Lisp and Functional Programming*, 1988.

[Rees86] Rees, J. A., and Clinger, William, Editors, *The Revised³ Report on the Algorithmic Language Scheme*, MIT/AI/Memo-848a, 1986.

[Reynolds78] Reynolds, J. C., "Definitional Interpreters for higher order programming languages", *Proceedings of the 5th Annual POPL*, January 1978, pages 39–46.

[Robinson65] Robinson, J. A., "A machine-oriented logic based on the resolution principle", JACM 12,1 (1965), pages 23–41.

[Tofte87] Tofte, Mads, *Operational Semantics and Polymorphic Type Inference*, Ph.D. Thesis, University of Edinburgh, 1987.

OFFICIAL DISTRIBUTION LIST

Director   2 copies
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research   2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627   6 copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center   12 copies
Cameron Station
Alexandria, VA 22314

National Science Foundation   2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38   1 copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555